

\* wizardzines.com \*

more at  
o this?



hello! we're here because bash\* is a very weird programming language.



\*most of this zine also applies to other shells, like zsh

## thanks for reading

There's more to learn about bash than what's in this zine, but I've written a lot of bash scripts and this is all I've needed so far. If the task is too complicated for my bash skills, I just use a different language.

two pieces of parting advice:

① when your bash script does something you don't understand, figure out why!

② use shellcheck! And read the shellcheck wiki when it tells you about an error :)

credits

Cover art: Vladimir Kašković  
Editing: Dolly Lanuza, Kamal Marhubi  
Copy Editing: Courtney Johnson  
and thanks to all 11 beta readers ♥

# debugging

26

## our hero: set -x

set -x prints out every line of a script as it executes, with all the variables expanded!

#!/bin/bash  
set -x → put set -x at the top

## or bash -x

\$ bash -x script.sh  
does the same thing as putting set -x at the top of script.sh

## you can stop before every line

trap read DEBUG  
↑  
the DEBUG "signal" is triggered before every line of code

## a fancy step debugger trick

put this at the start of your script to confirm every line before it runs:

```
trap '(read -p "[${BASH_SOURCE}:$LINENO] $BASH_COMMAND")' DEBUG  
↑  
read -p prints a script line  
message, press non-POSIX features..... 15  
enter to continue' → next command that will run
```

## how to print better error messages

this die function:

```
die() { echo $1 >&2; exit 1; }  
lets you exit the program and print a message if a command fails, like this:  
some_command || die "oh no!"
```

# table of contents

## basics

why I & bash .....	4
POSIX .....	5
shellcheck .....	6
variables .....	7
env variables .....	8
arguments .....	9
builtins .....	10
quotes .....	11
globs .....	12
redirects .....	13

## cheat sheets

(in the middle!)

\$()      \${}}	!!
{}      [:]	bash >
\$1      <	

if statements .....	16
for loops .....	17
reading input .....	18
functions .....	19
pipes .....	20
parameter expansion .....	21
background processes .....	22
subshells .....	23
trap .....	24
errors .....	25
debugging .....	26

## getting fancy

# Why I ❤️ Bash

4

**it's SO easy to get started**

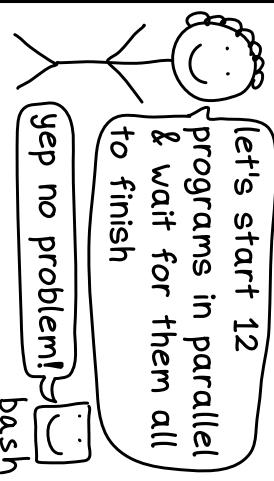
Here's how:

- ① Make a file called hello.sh and put some commands in it, like

ls /tmp

- ② Run it with bash hello.sh

**it's surprisingly good at concurrency**



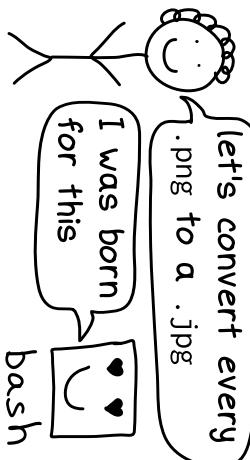
bash is weird and old, but the basics of how it works haven't changed in 30 years. If you learn it now, it'll be the same in 10 years.

managing pipes in other languages is annoying. in bash, it's just:

cmd1 | cmd2

**pipes & redirects are super easy**

**batch file operations are easy**



**bash is GREAT for some tasks**

But it's also EXTREMELY BAD at a lot of things. I don't use bash if I need:

- unit tests
- math (bash barely has numbers!)
- easy-to-read code 😊

## errors

**by default, bash will continue after errors**

that program's exit status was 1? who cares, let's keep running!!!

uh that is NOT what I wanted bash programmer

**variables don't error**

rm -r "\$HOME/\$SOMEPTH"

\$SOMEPTH doesn't exist? no problem, i'll just use an empty string!

OH NOOOO that means rm -rf \$HOME

**by default, a command failing doesn't fail the whole pipeline**

curl yxqzq.ca | wc

curl failed but wc succeeded so it's fine! success!

**set -e stops the script on errors**

set -e  
unzip file.zip  
script stops here.

this makes your scripts WAY more predictable

**set -u stops the script on unset variables**

set -u  
rm -r "\$HOME/\$SOMEPTH"

I've never heard of \$SOMEPTH!  
STOP EVERYTHING!!!

25

**set -o pipefail makes the pipe fail if any command fails**

you can combine set -e, set -u, and set -o pipefail into one command I put at the top of all my scripts:  
`set -euo pipefail`

# trap

24

**when your script exits, sometimes you need to clean up**

oops, the script created a bunch of temp files I want to delete

trap sets up callbacks

trap COMMAND EVENT  
what command to run  
when to run the command to run

**events you can trap**

- unix signals (INT, TERM, etc)
- the script exiting (EXIT)
- every line of code (DEBUG)
- function returns (RETURN)

trap COMMAND EVENT  
what command to run  
when to run the command to run

**example: Kill all background processes when Ctrl+C is pressed**

trap 'kill \$(jobs -p)' INT  
important: single quotes!  
when you press CTRL+C, the OS sends the script a SIGINT signal

trap COMMAND EVENT  
what command to run  
when to run the command to run

**example: cleanup files when the script exits**

```
function cleanup() {
    rm -rf $TEMPDIR
    rm $TEMPFILE
}
trap cleanup EXIT
```

trap COMMAND EVENT  
what command to run  
when to run the command to run

**bash runs COMMAND when EVENT happens**

trap "echo 'hi!!!'" INT  
<sends SIGINT signal>  
OS out 'hi!!!'  
ok, time to print

5

# POSIX compatibility

**there are lots of Unix shells**

dash zsh bash fish tcsh csh ksh

you can find out your user's default shell by running:

```
$ echo $SHELL
```

**POSIX is a standard that defines how Unix shells should work**

if your script sticks to POSIX, we'll all run it the same way! (mostly :)

I don't care about POSIX

**some shells have extra features**

we have extra features that aren't in POSIX

we keep it simple & just do what POSIX says

**on most systems, /bin/sh only supports POSIX features**

if your script has #!/bin/sh at the top, don't use bash-only features in it!

**this zine is about bash scripting**

most things in this zine will work in any shell, but some won't! page 15 lists some non-POSIX features

# Shellcheck

6

**shellcheck finds problems with your shell scripts**

\$ shellcheck my-script.sh  
oops, you can't use ~ in an if [ ... ]!  
Shellcheck

**it even tells you about misused commands**

hey, it looks like you're not using grep correctly here  
wow, I'm not! thanks!  
Shellcheck

**your text editor probably has a shellcheck plugin**

hey, that's a bash-only feature but your script starts with #!/bin/sh  
Shellcheck

**it checks for hundreds of common shell scripting errors (like "SC2013")**

and the shellcheck wiki has a page for every error, with examples! I've learned a lot from the wiki!

## subshells

**a subshell is a child shell process**

hey, can you run this bash code for me?  
sure thing! other bash process  
bash

**some ways to create a subshell**

- ① put code in parentheses (...) (cd \$DIR; ls)  
runs in subshell
- ② put code in \$(...)  
var=\$(cat file.txt)  
runs in subshell
- ③ pipe/redirect to a code block cat x.txt | while read line... piping to a loop makes the loop run in a subshell
- ④ + lots more for example, process substitution <() creates a subshell

**setting a variable in a subshell doesn't update it in the main shell**

```
cd subdir/  
mv x.txt y.txt  
) I like to do this so I don't have to remember to cd back at the end!
```

**it's easy to create a subshell and not notice**

```
x=$(some_function)  
(var=2)  
echo $var  
3, not 2  
this prints
```

I changed directories in some\_function, why didn't it work?  
it's running in a subshell!

# background processes

22

## scripts can run many processes in parallel

python -m http.server &  
curl localhost:8080  
& starts python in the "background", so it keeps running while curl runs

## wait waits for all background processes to finish

command1 &  
command2 &  
wait  
this waits for both command1 and command2 to finish

## background processes sometimes exit when you close your terminal

you can keep them running with nohup or by using tmux/screen.

```
$ nohup ./command &
```

## concurrency is easy\* in bash

in other languages:  
language: thing1 & thing2 & wait  
how do I do that again?  
\*(if you keep it very simple)

jobs, fg, bg, and disown let you juggle many processes in the same terminal, but I almost always just use multiple terminals instead

7

# variables

## how to set a variable

var=value<sup>right  
(no spaces!)</sup>  
var = value<sup>wrong</sup>  
var = value will try to run the program var with the arguments "=" and "value"

## how to use a variable

```
filename=blah.txt  
echo "$filename"  
they're case sensitive.  
environment variables are traditionally all-caps, like $HOME
```

## there are no numbers, only strings

a=2  
a="2"  
both of these are the string "2"  
technically bash can do arithmetic but I avoid it

## always use quotes around variables

\$ filename="swan 1.txt"  
\$ cat "\$filename"  
right!

prints "",  
zoo isn't a variable  
\$ zoo=panda  
\$ echo "\$zoo"  
\$ echo "\${zoo}2"  
"panda" like we wanted

wrong!  
\$ cat \$filename  
2 files!  
oh no!  
cat swan 1.txt  
mean that!  
um swan and 1.txt  
cat  
don't exist...

OK, I'll run  
cat "swan 1.txt"  
"swan 1.txt"!  
that's a file! yay!  
cat

# Environment variables

8

## every process has environment variables

printing out your shell's environment variables is easy, just run:

```
$ env
```

## child processes inherit environment variables

this is why the variables set in your .bashrc are set in all programs you start from the terminal. They're all child processes of your bash shell!

## shell scripts have 2 kinds of variables

1. environment variables
  2. shell variables
- unlike in other languages, in bash you access both of these in the exact same way: \$VARIABLE

## shell variables aren't inherited

var=panda  
 ↗  
 \$var only gets set in this process, not in child processes

**export sets environment variables**

how to set an environment variable:  
 export ANIMAL=panda  
 or turn a shell variable into an environment variable  
 ANIMAL=panda  
 export ANIMAL

## \${}: "parameter expansion" 21

`${{...}}` is really powerful

it can do a lot of string operations!  
 my favorite is search/replace.

`$[var]`

see page 7 for when to use this instead of \$var

`$[#var]`

length of the string or array var

`$[var#pattern]`

`$[var%pattern]`

/ replaces first instance,  
 // replaces every instance,  
 search & replace example:  
 \$ x="I'm a bearbear!  
 \$ echo {x/bear/panda}  
 I'm a pandabear!

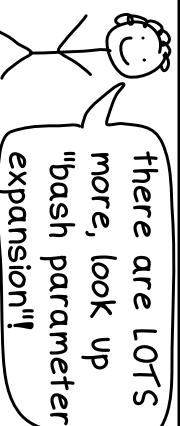
`$[var:offset:length]`

get a substring of var pattern from var. Example:

`$[var:-$othervar]`  
 use a default value like  
 \$othervar if var is unset/null

`$[var:?some error]`

prints "some error" and exits if var is unset/null

  
 there are LOTS more, look up "bash parameter expansion"!

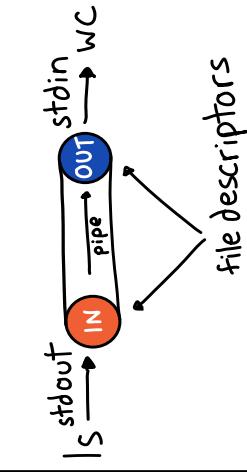
# Pipes

20

sometimes you want to send the output of one process to the input of another

```
$ ls | wc -l  
53 ↪ 53 files!
```

a pipe is a pair of 2 magical file descriptors



```
when ls does  
write(IN, "hi")  
wc can read it!  
read(OUT) → "hi"  
Pipes are one way. →  
You can't write to OUT
```

the OS creates a buffer for each pipe

IN (data waiting to be read)

when the buffer gets full:

write(IN, "...")

it's full! I'm going to pause you until process there's room again

```
OS
```

named pipes

you can create a file that acts like a pipe with mkfifo

```
$ mkfifo mypipe  
$ ls > mypipe &  
$ wc < mypipe } this does the same thing as ls | wc
```

only shell has the syntax process1 | process2 but you can create pipes in basically any language!

## arguments

Q

get a script's arguments with \$0, \$1, \$2, etc

```
$ svg2png old.svg new.png  
$0 is $1 is $2 is  
"svg2png" "old.svg" "new.png"  
(script's name)
```

arguments are great for making simple scripts  
Here's a 1-line svg2png script I use to convert SVGs to PNGs:

```
#!/bin/bash  
inkscape "$1" -b white --export-png="$2"  
I run it like this:  
$ svg2png old.svg new.png
```

"\$@": all arguments

\$@ is an array of all the arguments except \$0.  
This script passes all its arguments to ls --color:

```
#!/bin/bash  
ls --color "$@"
```

shift removes the first argument

this prints the script's first argument

```
echo $1 ← shift  
echo $1 ← shift  
echo $1 ← shift
```

this prints the second argument

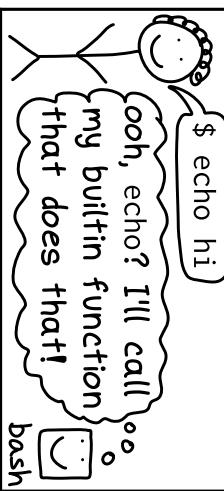
# builtins

10

## most bash commands are programs

You can run `which` to find out which binary is being used for a program:

```
$ which ls
/bin/ls
```



## examples of builtins

`type`    `source`  
`alias`    `declare`  
`read`    `cd`  
`printf`    `echo`

type tells you if a command is a builtin

```
$ type grep
grep is /bin/grep
$ type echo
echo is a builtin
$ type cd
cd is a builtin
```

## a useful builtin: alias

alias lets you set up shorthand commands, like:  
`alias gc="git commit"`  
`~/.bashrc` runs when bash starts, put aliases there!

## a useful builtin: source

bash script.sh runs script.sh in a subprocess, so you can't use its variables / functions.  
source script.sh is like pasting the contents of script.sh

# functions

## defining functions is easy

```
say_hello() {
    echo "Hello!"
```

... and so is calling them  
`say_hello` ↪ no parentheses!

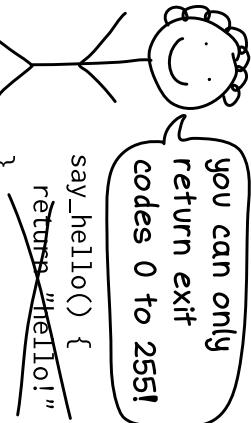
## functions have exit codes

```
failing_function() {
    return 1
}
```

0 is success, everything else is a failure. A program's exit codes work the same way.

## you can't return a string

you can only return exit codes 0 to 255!

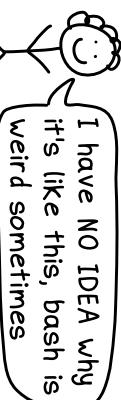


## the local keyword declares local variables

```
say_hello() {
    echo "Hello $1!"
}
say_hello "Ahmed"
not say_hello("Ahmed")!
```

## local x=VALUE suppresses errors

local x=\$asdf → never fails, even if asdf doesn't exist  
`local x` → this one will fail  
`x=$(date)` → local  
`y=$(date)` → global



# reading input

18

**read -r var  
reads stdin into  
a variable**

\$ read -r greeting  
hello there! ← type here  
\$ echo "\$greeting" and press  
enter  
hello there!

**you can also read  
into multiple variables**

```
$ read -r name1 name2  
ahmed fatima  
$ echo "$name2"  
fatima
```

**by default, read  
strips whitespace**

```
" a b c " -> "a b c"  
it uses the IFS ("Input  
Field Separator") variable  
to decide what to strip
```

**set IFS=''** to avoid  
stripping whitespace

```
$ IFS='', read -r greeting  
hi there!  
$ echo "$greeting"  
hi there!  
~~~~~  
the spaces are  
still there!
```

**more IFS uses: loop over every line of a file**  
by default, for loops will loop over every word of a file  
(not every line). Set IFS=''' to loop over every line instead!  
IFS=''  
→ for line in \$(cat file.txt)  
do  
  echo \$line  
done  
→ don't forget  
to unset IFS  
when you're  
done!

## quotes

||

**double quotes expand variables,  
single quotes don't**

```
$ echo 'home: $HOME'  
home: $HOME  
→ $HOME got expanded  
to /home/bork  
single quotes always  
give you exactly what  
you typed in
```

**you can quote  
multiline strings**

```
$ MESSAGE="Usage:  
here's an explanation of  
how to use this script!"
```

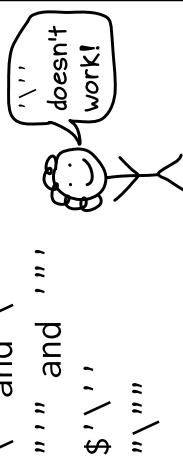
**how to concatenate  
strings**

put them next to each other!  
\$ echo "hi ""there"  
hi there  
x + y doesn't add strings:  
\$ echo "hi" ± " there"  
hi ± there

**a trick to escape  
any string: !:q:p**

```
get bash to do it for you!  
$ # He said "that's $5"  
$ !:q:p  
' # He said "that'\''s $5"  
this only works in bash, not zsh.  
! is an "event designator" and  
:q:p is a "modifier"
```

**escaping ' and "**  
here are a few ways  
to get a ' or ':  
\' and \"  
''' and ''',  
\$'\'',  
"\\" doesn't  
work!

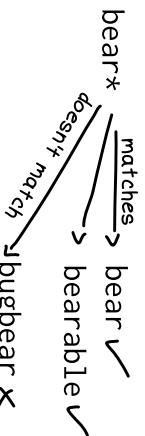


# globs

12

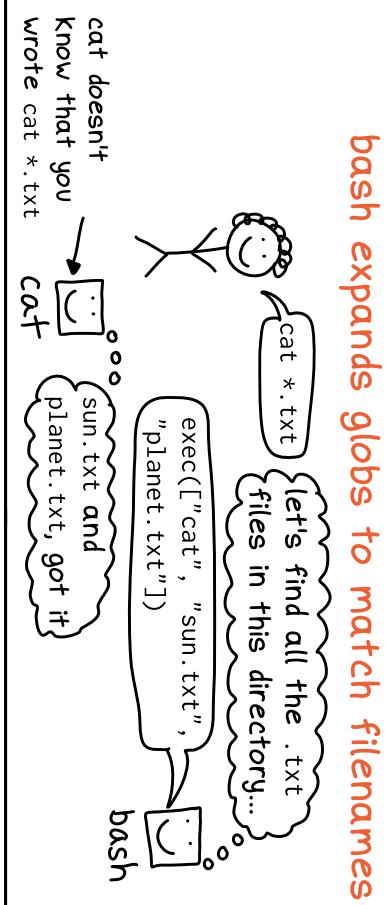
## globs are a way to match strings

beware: the \* and the ? in a glob are different than \* and ? in a regular expression!!!



## there are just 3 special characters

- \* matches 0+ characters
- ? matches 1 character
- [abc] matches a or b or c



# for loops

## for loop syntax

```
for i in panda swan
```

```
do  
    echo "$i"  
done
```

for loops loop over words, not lines

```
for word in $(cat file.txt)
```

loops over every word in the file, NOT every line (see page 18 for how to change this!)

## the semicolons are weird

usually in bash you can always replace a newline with a semicolon. But not with for loops!

for i in a b; do ...; done  
you need semicolons before do and done but it's a syntax error to put one after do

## looping over files is easy

```
for i in *.png
```

```
do  
    convert "$i" "${i/png/jpg}"  
done
```

this converts all png files to jpgs!

## how to loop over a range of numbers

```
while COMMAND
```

```
do  
    ...  
done
```

like an if statement, runs COMMAND and checks if it returns 0 (success)

```
3 ways:  
for i in $(seq 1 5)  
for i in {1..5}  
for ((i=1; i<6; i++))
```

these two only work in bash, not sh

# if statements

16

in bash, every command has an **exit status**

- 0 = success
- any other number = failure

bash puts the exit status of the last command in a special variable called `$?`:

**why is 0 success?**

there's only one way to succeed, but there are lots of ways to fail. For example  
grep THING x.txt will exit with status:  
1 if THING isn't in x.txt  
2 if x.txt doesn't exist

**bash if statements test if a command succeeds**

```
if COMMAND; then
    # do a thing
fi:  
this:  
① runs COMMAND  
② if COMMAND returns 0 (success), then do the thing
```

**[ vs [**

there are 2 commands often used in if statements: [ and [[  
if [ -e file.txt ] if [[ -e file.txt ]]  
/usr/bin/[ (aka test) is [[ is built into bash. It  
a program\* that returns treats asterisks differently:  
0 if the test you pass it [[ \$filename = \*.png ]]  
succeeds doesn't expand \*.png into  
\*in bash, [ is a builtin that acts like /usr/bin/[

**true**  
true is a command that always succeeds, not a boolean

**combine with && and ||**  
if [ -e file1 ] && [ -e file2 ]  
**man test for more on [**  
you can do a lot!

> redirects <

13

**unix programs have 1 input and 2 outputs**

When you run a command from a terminal, they all go to/from the terminal by default.

**< redirects stdin**  
\$ wc < file.txt  
\$ cat file.txt | wc }  
these both read file.txt to wc's stdin each input/output has a number (its "file descriptor")

**> redirects stdout**  
\$ cmd > file.txt  
\$ cmd > /dev/null

**>> redirects stderr**  
\$ cmd >> file.txt  
\$ cmd >> /dev/null

**2>&1 redirects stderr to stdout**

```
$ cmd > file.txt 2>&1
$ cmd > /dev/null
$ cmd >> file.txt
$ cmd >> /dev/null
```

**/dev/null**

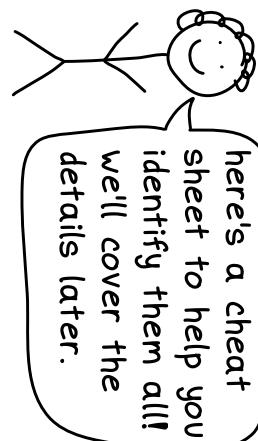
your operating system ignores all writes to /dev/null

**sudo doesn't affect redirects**  
your bash shell opens a file to redirect to it, and it's running as you. So  
\$ sudo echo x > /etc/xyz won't work. do this instead:  
\$ echo x | sudo tee /etc/xyz

# brackets cheat sheet

14

**shell scripts have a lot of brackets**



```
x=$((2+2))
```

`$(( ))` does arithmetic

`a{.png,.svg}`  
this expands to `a.png a.svg`  
it's called "brace expansion"

`(cd ~/music; pwd)`  
`(...)` runs commands in a subshell.

`{ cd ~/music; pwd }`  
`{...}` groups commands.  
`x=(1 2 3)`  
`x=(...)` creates an array

```
if [ ... ]
```

`/usr/bin/[` is a program  
that evaluates statements

```
if [[ ... ]]
```

`[[` is bash syntax. it's more powerful than `[`

```
<((COMMAND))
```

"process substitution":  
an alternative to pipes

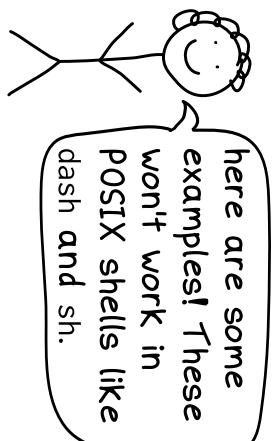
```
$(var//search/replace}
```

see page 21 for more  
about `$(...)`!

## non-POSIX features

15

**some bash features aren't in the POSIX spec**



```
[[ ... ]]
```

POSIX alternative:  
`[ ... ]`

```
diff <(./cmd1) <(./cmd2)
```

this is called "process substitution", you can use named pipes instead

```
{1..5}
```

POSIX alternative:  
`$[seq 1 5]`

```
$'\\n'
```

POSIX alternative:  
`$(printf "\n")`

```
$(var//search/replace}
```

**arrays**  
POSIX shells only have one array: `$@` for arguments

```
[[ $DIR = /home/* ]]
```

POSIX alternative:  
match strings with grep

`VAR=$(cat file.txt)`  
`$COMMAND` is equal to COMMAND's stdout

`$(var//search/replace)`  
POSIX alternative: pipe to sed